Unity Catalog: Open and Universal Governance for the Lakehouse and Beyond

Ramesh Chandra
Haogang Chen
Ray Matharu
Sarah Cai
Jeff Chen
Priyam Dutta
Bogdan Ghita
Todd Greenstein
Gopal Holla
Peng Huang
Yuchen Huo
Adrian Ionescu
Adriana Ispas
Databricks
San Francisco, CA, USA

Tim Januschowski
Vihang Karajgaonkar
Stefania Leone
David Lewis
Andrew Li
Nong Li
Cheng Lian
Stephen Link
Qing Lu
Yesheng Ma
Chris Pettitt
Vijayan Prabhakaran
Databricks
San Francisco, CA, USA
firstname.lastname@databricks.com

Bogdan Raducanu
Kyle Rong
Paul Roome
Samarth Shetty
Sean Smith
Xiaotong Sun
Yuyuan Tang
Weitao Wen
Lei Xia
Junlin Zeng
Ben Zhang
Reynold Xin
Matei Zaharia
Databricks
San Francisco, CA, USA

Abstract

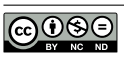
Enterprises are increasingly adopting the Lakehouse architecture to manage their data assets due to its flexibility, low cost, and high performance. While the catalog plays a central role in this architecture, it remains underexplored, and current Lakehouse catalogs exhibit key limitations, including inconsistent governance, narrow interoperability, and lack of support for data discovery. Additionally, there is growing demand to govern a broader range of assets beyond tabular data, such as unstructured data and AI models, which existing catalogs are not equipped to handle. To address these challenges, we introduce Unity Catalog (UC), an open and universal Lakehouse catalog developed at Databricks that supports a wide variety of assets and workloads, provides consistent governance, and integrates efficiently with external systems, all with strong performance guarantees. We describe the primary design challenges and how UC's architecture meets them, and share insights from usage across thousands of customer deployments that validate its design choices. UC's core APIs and both server and client implementations have been available as open source since June 2024.

CCS Concepts

• **Information systems** → **Data management**; • **Security and privacy** → **Database and storage security**.

Keywords

Metadata Management, Data Governance, Lakehouse Catalog



SIGMOD-Companion '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724459

ACM Reference Format:
Ramesh Chandra, Haogang Chen, Ray Matharu, Sarah Cai, Jeff Chen, Priyam Dutta, Bogdan Ghita, Todd Greenstein, Gopal Holla, Peng Huang, Yuchen Huo, Adrian Ionescu, Adriana Ispas, Tim Januschowski, Vihang Karajgaonkar, Stefania Leone, David Lewis, Andrew Li, Nong Li, Cheng Lian, Stephen Link, Qing Lu, Yesheng Ma, Chris Pettitt, Vijayan Prabhakaran, Bogdan Raducanu, Kyle Rong, Paul Roome, Samarth Shetty, Sean Smith, Xiaotong Sun, Yuyuan Tang, Weitao Wen, Lei Xia, Junlin Zeng, Ben Zhang, Reynold Xin, and Matei Zaharia. 2025. Unity Catalog: Open and Universal Governance for the Lakehouse and Beyond. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3722212.3724459

1 Introduction

Many enterprises store their data in elastic data lakes such as Amazon S3, Azure Data Lake Storage, and Google Cloud Storage (see e.g., [27] for an overview), and are increasingly adopting the Lakehouse architecture [35], which streamlines data management relative to traditional data warehouses by bringing data warehouse capabilities directly to data lakes. This enables users to process all their data in a uniform way, with a variety of compute engines.

Despite its popularity, a critical yet understudied component of the Lakehouse architecture is the catalog. Current Lakehouses typically rely on table catalogs like the Hive Metastore (HMS) [11] as the operational catalog to organize data assets into a namespace and manage their metadata. However, existing catalogs exhibit several limitations. First, they provide limited and inconsistent governance. For instance, access enforcement can differ depending on whether a table is accessed via its catalog name or through its cloud storage path, without a mechanism to enforce consistency—a key challenge when presenting a table abstraction over data lake storage systems that expose separate APIs. Moreover, many catalogs

lack support for fine-grained access control features such as view-based access control or row- and column-level security. Second, while HMS is open and interoperable, many proprietary catalogs, including AWS Glue [7] or those tightly integrated with specific data warehouses, are not open, making it difficult to access data from external workloads. Finally, existing operational catalogs often lack discovery capabilities (e.g., searching for data tagged as 'PII' or tracking data lineage), which are critical for many users.

Furthermore, enterprises increasingly seek to manage and govern a broad range of assets beyond tables, including multiple versions of AI models, unstructured data (e.g., images and text), and even on-premise data sources, in a unified way. The Lakehouse architecture offers a natural solution for storing these assets in low-cost data lake storage. However, existing data lake catalogs are primarily designed for tabular data and lack the capability to represent unstructured data or AI assets, rendering them inadequate for the increasingly important AI/ML workloads.

To address these limitations, an ideal Lakehouse catalog should meet these fundamental requirements:

- **Consistent governance.** Ensures uniform enforcement of access policies across all access paths, while offering comprehensive auditing and fine-grained access control capabilities.
- **Universality.** Extensible to support diverse asset types (e.g., tabular, unstructured, AI assets), interoperates with various clients (e.g., SQL engines, User Interfaces (UI), AI tools), provides both operational and discovery catalog features, enables cross-organizational sharing, and operates across multiple cloud environments.
- **High Performance.** Supports workloads with varying performance and consistency requirements, including low-latency interactive workloads, transactional workloads requiring strong consistency, and high-throughput batch workloads.

This paper presents Unity Catalog (UC), an open Lakehouse catalog developed at Databricks to address these requirements. At a high-level, UC provides a unified namespace for Lakehouse assets, and introduces an entity-relationship data model that is extensible to a broad range of data and AI asset types. It includes built-in governance capabilities for these assets, and an API that enables client integration to deliver end-to-end functionality to users.

The core UC API, along with its server and client implementations, has been open-sourced since June 2024 and currently supports key asset types such as tables and ML models. In Databricks, UC serves as the central catalog across all product features, with additional asset types and platform-specific optimizations implemented through UC's extension points. UC has been running in production at Databricks since 2021 and is actively used by ~9,000 customers, managing ~100 million tables, ~550,000 volumes, and ~400,000 ML models, and serving ~60,000 API calls per second.

Designing UC required addressing the following key challenges.

Uniform access control. Lakehouse workloads access tables at two levels of abstraction: as higher-level catalog assets or directly through cloud storage paths. Existing catalogs, like HMS, operate solely at the catalog level, and direct access via raw storage paths bypasses the catalog entirely. However, customers require consistent access control regardless of the access method. UC addresses this challenge with two key mechanisms: (i) it enforces a *one-asset-per-path* principle, so that each cloud storage object maps to at most one UC asset; (ii) clients do not have direct access to cloud storage; instead, UC provides a *credential vending* API that issues temporary credentials to clients. When a path-based access request is made, UC resolves the asset from the path and enforces its access policies, before issuing temporary credentials to that asset's storage.

Support for diverse asset types. Customers need to manage assets beyond tables for modern data and AI workloads, and also wish to manage assets in external catalogs like HMS from UC. To serve as a universal catalog, UC enables this through two key mechanisms. First, it provides an open, extensible API to integrate various open formats as asset types. For example, making UC act as a MLflow [34] model registry only involved creating UC implementations of the MLflow base model registry `RestStore` and the base `ArtifactRepository`, after adding UC asset type for registered models. UC uses the same one-asset-per-path and credential vending approaches to ensure uniform governance for the model asset type. Second, UC implements *catalog federation*, which allows users to "mount" data managed by an external catalog, such as an on-premise DBMS or HMS, and make it accessible in UC.

External access. An important use case is sharing data with external workloads that may not support the format of data stored in UC, without requiring additional data copies. For instance, this includes scenarios like sharing a Delta Lake [6] table with an external workload that does not understand the Delta format or with one that only supports Apache Iceberg [3]. UC addresses this challenge by providing multiple open interfaces to access the same underlying data, including the Delta Sharing protocol [15] to share Delta tables with external Delta Sharing clients, Delta UniForm (Universal Format) [16] to allow external Iceberg and Hudi clients to read Delta tables in UC, and the Iceberg REST Catalog interface [24] to provide access to the UC catalog functionality to Iceberg clients.

Discovery support. Many enterprise use cases require identifying assets that meet specific criteria or understanding their lifecycles. For instance, a user may need to verify that an asset has no downstream dependencies—using lineage information—prior to deletion, or may wish to locate all assets tagged with 'PII'. These needs are typically addressed by *discovery catalogs*, which operate by "indexing" metadata from operational catalogs. However, separating operational and discovery catalog functionality introduces several challenges: (i) collecting metadata like lineage requires coordination not only with the operational catalog but also with compute engines; (ii) discovery catalogs often rely on polling operational catalogs for updates, which incurs overhead and necessitates tradeoffs between metadata freshness and system performance; (iii) efficiently enforcing access control policies defined in the operational catalog during discovery queries adds additional complexity.

UC is designed with discovery catalog functionality in mind to address these challenges. It provides lineage APIs that allow compute engines to submit lineage information for end-to-end data tracking. To improve freshness and reduce polling overhead, UC offers a change event stream that allows discovery catalogs to receive timely updates. Additionally, UC exposes authorization APIs that enable discovery systems to efficiently enforce access control policies at query time. In Databricks, these capabilities power features such as lineage and search, and the same functionality is exposed via APIs to enterprise discovery platforms like Collibra and Alation.

Performance. Data warehouse workloads require Lakehouse catalogs to deliver significantly lower latency than traditional data lake solutions. UC operates as an independent service that exposes an open API, enabling integration with a variety of compute engines. This separation is essential for UC to function as a universal catalog, but it introduces additional network hops between engines and the catalog service, which can increase metadata access latency. To mitigate this overhead, UC incorporates both batching and caching mechanisms. It consolidates all metadata access for a query into a single batched API call, and employs specialized caches based on the consistency requirements of the metadata. For immutable metadata or metadata where weak consistency is acceptable (e.g., cloud credentials or user/group information), UC uses simple TTL-based caches to bound staleness. For metadata requiring stronger consistency—such as table commit information—UC utilizes a write-through cache that respects the transaction isolation guarantees of its underlying metadata store. These caches can be pushed to clients to further reduce latency for frequently accessed metadata.

The rest of the paper is organized as follows. Section 2 presents related work, Section 3 gives an overview of UC to set context for Section 4, which discusses how key aspects of UC's design address the above challenges. Section 5 highlights implementation details, and Section 6 presents UC's usage statistics and evaluation results.

2 Background and Related Work

Catalogs come under the broader umbrella of metadata management, which has been studied theoretically (e.g., [8, 25]) and in applied work (e.g., [19–21]). Despite their practical importance and existing proprietary and open source implementations for different data management systems, catalogs remain an understudied area.

To differentiate the existing approaches, we contrast traditional relational database systems (DBMS) with modern Lakehouses. In a traditional DBMS, the catalog is a key component of a monolithic architecture (e.g., [18, 29]) that stores schema metadata, such as details about tables, columns, and relationships, and is tightly coupled with query processing and data organization. While this tight integration simplifies implementing end-to-end functionality spanning metadata and data, including optimizations and maintaining consistency between them, it limits the scope and flexibility as the catalog is bound to a specific engine and storage format.

In contrast, the Lakehouse architecture [35] separates data storage from query engines to achieve greater flexibility and scalability. By decoupling them, Lakehouses allow independent scaling of storage and compute resources, to better handle big data workloads. This also enables the use of specialized engines, each optimized for different workloads—such as BI, ETL, or ML training—while operating on the same copy of data. The Lakehouse catalog is also separated from the engines and data storage, and this separation of data storage, catalog, and engines makes it critical for organizations to ensure consistent governance across the workloads.

Arguably the most widely used Lakehouse catalog today is the Hive Metastore [11], an open-source project used by many open source engines. Proprietary catalogs like AWS Glue [7] and managed HMS offerings like Dataproc Metastore [13] also provide a HMS compatible interface for interoperability with existing engines. HMS provides the basic catalog functionality of organizing tables into schemas and storing metadata for each table, and clients use it to retrieve table metadata including the location and then access cloud storage to process data. This makes it simple for clients to integrate with HMS, but HMS does not have governance (it relies on cloud storage policies for access control), does not natively support advanced data warehouse functionality like transactions, and does not support asset types beyond tables.

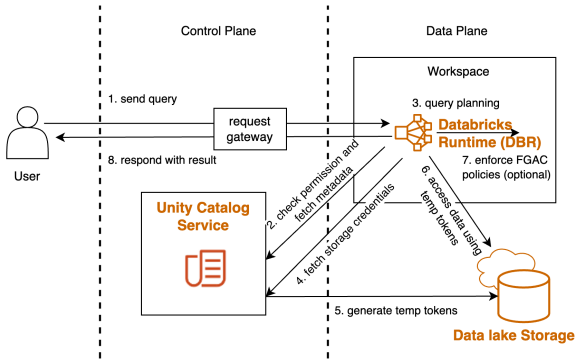


Figure 1: The life of a SQL query in the Databricks Lakehouse platform, which illustrates the main components of the platform, separation of the engine and the catalog, and the data and control planes. Arrows represent the logic flow.

The recent Iceberg REST Catalog API [24] is another increasingly adopted open catalog API. It is specific to the Iceberg table format and supports transactions, but does not specify governance APIs beyond temporary credential vending and does not support assets beyond Iceberg tables. Polaris [4] is an open source implementation of the Iceberg REST Catalog API and adds on a concrete governance API, but it too does not support assets beyond Iceberg tables.

Commercial offerings include Fabric/Purview/OneLake [28], AWS Glue, BigLake/Dataplex [9], and Horizon [31]. These catalogs are not open (though some of them, like AWS Glue, provide a HMS or Iceberg REST catalog interfaces for interoperability), and they do not support managing assets beyond tables.

In contrast to the existing catalogs, Unity Catalog is open; has built-in consistent governance including access control, temporary credential vending for underlying cloud storage, and auditing; supports data and AI assets beyond tables; and is designed with external access and discoverability in mind.

3 Unity Catalog Overview

This section provides an overview of UC, beginning with a brief primer on the Databricks Lakehouse architecture to establish context. It then introduces UC's object and permission models, followed by a summary of the life cycle of a SQL query, highlighting key design elements that are discussed in greater detail in Section 4.

3.1 Databricks Lakehouse Platform

The Databricks Lakehouse platform consists of three key components relevant to UC's design (illustrated in Figure 1): the data lake storage, the Databricks runtime, and the Unity Catalog service.

Data lake storage. The Lakehouse platform decouples storage from compute, allowing customers to choose their preferred storage

providers (e.g., S3, ADLS, GCS) and bring existing large datasets without costly migrations. These datasets can be stored in open formats such as Delta Lake, Iceberg, or Parquet.

Databricks Runtime (DBR). The Databricks Runtime is the core execution engine that powers data processing across a wide range of workloads, including SQL, machine learning, and both interactive and batch operations. Built on open-source Apache Spark [36], it incorporates substantial enhancements in performance and reliability. The engine is structured around *clusters*, each comprising a driver node responsible for query planning and one or more executor nodes that execute tasks in parallel. Clusters are associated with *workspaces*, which offer a unified environment for teams, integrating compute resources, notebooks, jobs, and other assets.

Unity Catalog service. The Unity Catalog service is a multi-tenant service implementing all UC functionality and APIs. Execution engines interact with it to deliver end-to-end functionality for users. The Unity Catalog service is covered in more detail in Section 4.2.1.

The data lake storage and DBR are part of the *data plane*. Customers have the option to manage the cloud resources for the data plane themselves within their cloud accounts or to use Databricks-managed ones within a Databricks-controlled cloud account. The Unity Catalog service operates in the *control plane*, which runs entirely within a Databricks-managed cloud account and is fully operated by Databricks.

3.2 Object Model

The UC object model is shown in Figure 2. All data and AI assets in UC are organized within a metastore, which defines a three-level hierarchical namespace. Assets are referenced using fully qualified names of the form "catalog.schema.table". The first two levels, namely catalogs and schemas, are containers that provide different levels of logical isolation required by enterprises. The last level contains the actual assets such as tables, views, volumes, ML models, and functions. Common functionality across these asset types is abstracted into a generic entity-relationship data model, which is described in more detail in Section 4.2.1.

Catalogs typically reflect organizational units (e.g., per team) or development scopes (e.g., separate catalogs for development and production). Administrators can define "bindings" to restrict a catalog's access to specific Databricks workspaces. Catalogs also serve as integration points for importing external data via Delta Sharing and federation.

Schemas (or databases) reside within catalogs and organize data and AI assets into finer-grained categories. A schema often represents a use case, project, or team sandbox, with access isolation enforced through privileges. Alongside traditional asset types like tables, views, and functions, UC also supports volumes, which represent a logical storage in a cloud object storage location for organizing files and non-tabular data, and models, which represent ML models and their associated artifacts.

A metastore serves as the root of the namespace and provides the highest level of naming isolation. Each metastore is associated with a "home region," where its primary metadata is stored. Every Databricks workspace—and its workloads—is attached to a single metastore, and objects in other metastores are inaccessible by default unless explicitly shared via Delta Sharing [15]. To optimize for locality, workspaces attach to a metastore in the same region by default. In addition to catalogs, a metastore also contains non-data configuration assets that abstract cloud resources, including credentials, storage locations, and connections, which abstract cloud principals, cloud storage, and external data sources, respectively. These assets enable UC to operate across clouds and engines.

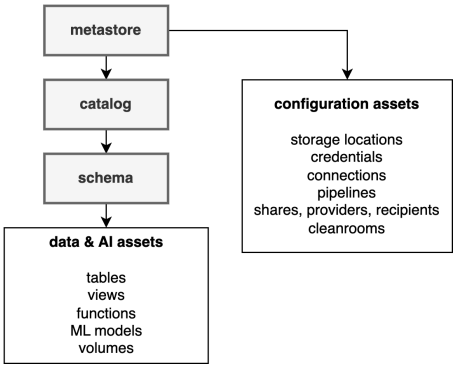


Figure 2: The Unity Catalog object model, showing its three-level hierarchical namespace and common asset types.

3.3 Privilege Model

In UC, users require specific privileges to perform operations on objects. UC's privilege model is inspired by SQL-style grants. For example, to run a SELECT query on a table, a user must have the SELECT privilege on the table, the USE SCHEMA privilege on its parent schema and the USE CATALOG privilege on its catalog; the usage privileges allow container administrators to enforce broad access restrictions on assets within the container, similar to other SQL databases. Principals can be granted privileges on any object within a metastore. These objects (containers, assets or configurations) are referred to as *securables* in this section. Below are the key aspects of UC's privilege model.

Ownership. Every securable has an owner who holds all privileges on the object, including administrative rights, and can grant privileges to other principals. By default, regular users have no access to any securables within a metastore. Administrators must bootstrap access by granting privileges, including the right to create new securables. When a securable is created, its creator is automatically assigned as its owner. Owners may delegate privilege management by granting the MANAGE privilege to other principals, which confers the same authority as ownership.

Privilege inheritance. Privileges in UC are inherited down the securable hierarchy. Granting a privilege on a catalog or schema automatically propagates that privilege to all existing and future securables within that scope. For example, a principal granted SELECT on a catalog receives SELECT on all current and future tables in that catalog. This inheritance model allows administrators to scale access control efficiently by leveraging hierarchical grants.

Administrative privileges are also inherited, enabling container administrators to manage privileges for all descendant securables. However, they do not receive non-administrative privileges by default. For instance, a schema owner does not automatically gain SELECT access to tables in the schema unless they explicitly grant it

to themselves. This separation is crucial in regulated environments, as it prevents unintended data access by privileged users.

Fine-grained access control (FGAC). Some use cases require restricting access to specific rows or columns within a table—for example, limiting access to sensitive data such as social security numbers for non-privileged users. UC enables table administrators to define row filters and column masks, allowing data to be selectively hidden based on the accessing principal. Implementing fine-grained access control (FGAC) requires coordination between UC and a "trusted" query engine, as discussed further in Section 4.3.2.

Attribute-based access control (ABAC). To support scalable access management in large enterprises, administrators often prefer to define high-level access control policies based on data attributes rather than individual securables. ABAC enables this by dynamically applying privilege grants or FGAC policies based on metadata—such as tags—associated with a securable or table column. For example, an administrator can define an ABAC policy at the catalog level that applies a redacting column mask to all columns tagged with 'PII' for unprivileged users. An ABAC policy applies to all current and future securables within the policy's scope that satisfy the specified conditions. This approach enables flexible, comprehensive, and hierarchical policy enforcement at scale. ABAC is currently available in private preview.

3.4 Life of a SQL Query

This section gives an overview of the interaction of DBR and Unity Catalog during the execution of a typical SQL query to highlight UC's key design elements. Figure 1 illustrates this interaction.

(1) When a DBR engine receives a user's SQL query (via a request gateway that authenticates the user), it first parses the query and finds all securable references, such as table names.
(2) *Metadata resolution and access control*: The engine issues REST API requests to UC to retrieve metadata for the requested securables. UC first verifies that the caller has the necessary privileges and then returns metadata such as column definitions and constraints for a table. For composite securables, such as views and functions, UC also performs dependency resolution, authorizing access and including metadata for all referenced securables. If the table is subject to FGAC (e.g., row filters) and the engine is trusted, the response includes the applicable enforcement rules. Details on securable-level access control and FGAC are provided in Sections 4.3.1 and 4.3.2, respectively.
(3) The engine analyzes the query and generates a query plan using the returned metadata. A large query might be divided into smaller tasks and distributed to executors for processing.
(4) When the engine requires access to cloud storage that hosts the table data, it sends API requests to UC to fetch a short-lived credential with the appropriate access level.
(5) *Credential vending*: upon authorizing the credential request, UC returns a least-privileged access token that is downscoped to only have access to the data asset in question. UC might cache unexpired tokens to accelerate future access. Credential vending is described in Section 4.3.1.
(6) *Storage access*: the engine executors access cloud storage directly using the received access token to process the query.
(7) (Optional) For tables subject to FGAC, a trusted engine applies filtering or transformation to the query results based on the enforcement rules included in the table metadata.
(8) The engine sends the final query result back to the user.

Audit logging and lineage tracking are done during query processing in DBR and the Unity Catalog service; these are covered in Section 4.2.1. UC is designed to be open, so the above interaction has to generalize to other engines beyond DBR; the principle of catalog-engine separation is discussed in Section 4.1.

4 System Design

This section describes the key components of UC's design that address the challenges from Section 1. Some functionality described here is not yet available in the UC open-source implementation. This may be because it is planned for future open-sourcing, relies on a missing open standard (e.g., a "trusted engine" standard for FGAC), or depends on Databricks internal infrastructure (e.g., caching).

4.1 Catalog-Engine Separation

As shown in Figure 1, the Lakehouse architecture follows the principle of catalog-engine separation, even though workloads often require close collaboration between the catalog and a processing engine. Here, the term *engine* refers broadly to any client that processes not only metadata but also data. This includes tabular engines like Trino [32] and Apache Spark [36], as well as ML clients that perform training. UC defines a clear interface between the catalog and the engines.

The separation between the catalog and engines provides two major benefits. First is enhanced manageability and security. The catalog acts as a centralized authority for the Lakehouse asset namespace, all asset metadata, and access control policies. This single source of truth ensures that the core metadata is tightly managed, with well-defined REST APIs. The namespace and governance are unambiguous across different workloads and engines. Metadata operations can be properly ordered if necessary to support use cases like commit protocols on the catalog assets.

Second, it improves interoperability across engines. A unified catalog allows different engines to work with the same set of assets, making it easier to support a wide variety of workloads without duplicating metadata management logic in each engine. Engines can focus on processing logic while relying on the catalog for metadata and access control, which improves modularity, reduces duplication of effort, and allows users to use the best engine for their workload. Since the catalog does not serve or process data, it is agnostic to the underlying format for the assets it manages. This gives the engine full flexibility to choose and optimize the data layout, for example, engines can choose optimizations such as Delta Lake deletion vectors [14], without the catalog getting in the way.

Conversely, the catalog API by itself is not enough to provide the functionality users need. Various engines need to use the catalog API in the right way to provide the end-to-end functionality. For instance, in the SQL query example in Section 3.4, FGAC requires a trusted engine to correctly interpret and faithfully enforce the row filters or column masks in order to meet the end-to-end access control requirements. Other UC features that require catalog-engine collaboration include fine-grained lineage tracking (see Section 4.4),

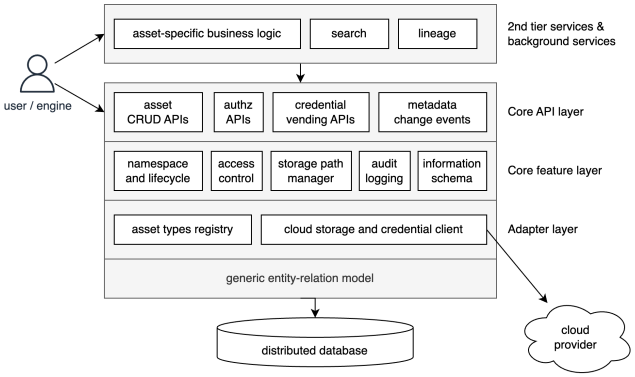


Figure 3: The layered architecture of the Unity Catalog service, showing the separation of the functionality core to all asset types and functionality specific to an asset type, and the distinction between the foreground and background services.

multi-table transaction handling (see Section 6.3), external catalog federation (see Section 4.2.4) and view materialization.

4.2 Support for Diverse Asset Types

This section describes the UC functionality that supports diverse data and AI asset types to make it a universal catalog. It begins with an overview of the Unity Catalog service, followed by a discussion of the generic entity-relationship data model that serves as the foundation for all asset types. Next, it illustrates how this was used to incorporate ML model support into UC. Finally, it explains how UC's catalog federation imports data from external catalogs.

4.2.1 Unity Catalog Service. A key objective of UC's design is to maintain a uniform behavior for the common feature set shared by all asset types, while allowing different asset types to have type-specific functionalities. The Unity Catalog service, whose architecture is shown in Figure 3, implements the core UC functionality. The following metadata and functionality served by the Unity Catalog service is considered essential for all asset types.

Asset namespaces. UC maintains the names of all assets and enforces the uniqueness of fully qualified names for each asset type. For example, it enforces that two table-like assets (e.g., a table and a view) cannot have the same name in a given schema.

Storage paths and the *one-asset-per-path* principle. An asset with storage, like a table or a volume, can either be *managed* or *external*, depending on who allocates its cloud storage paths. For managed assets, the Unity Catalog service allocates and de-allocates cloud storage paths, whereas for external assets, storage path management is done outside, with UC only storing the metadata for the assets. In both cases, the catalog enforces the one-asset-per-path principle, which dictates that no two assets in a metastore can have overlapping storage paths. This invariant guarantees an unambiguous mapping from a cloud storage path to the corresponding UC asset and rules out ambiguous (or conflicting) access control policies when the same data is accessed through different names.

Lifecycle. The Unity Catalog service keeps track of the creation and deletion of assets in the 3-level namespace. It handles soft deletions and propagates deletions of a parent object to its children. It also garbage collects orphaned objects and cleans up the associated resources, such as the storage blob for managed tables.

Access control. The Unity Catalog service maintains object ownership, privilege grants, tag assignment, ABAC rules and fine-grained access control policies. It is also the sole authority to make access control decisions based on these governance metadata.

Credentials. Credentials used to access cloud storage and external data sources are secured and governed by the Unity Catalog service. It is also responsible for generating temporary down-scoped credentials for data assets based on the requested access level.

Audit logging. The Unity Catalog service maintains an audit trail for API requests, object life cycle changes, access control decisions and other important events for all asset types.

4.2.2 Entity-Relationship Data Model. UC's layered design, shown in Figure 3, is essential for the separation of functionality core to all asset types and functionality specific to certain asset types.

At the bottom there is a generic entity-relationship data model, which is the building block for metadata operations of all asset types. The model implements common interfaces such as asset look up by name or by ID, parent-child look up and listing, retrieval of privilege grants, and the state machine for resource provisioning and clean up. The model also provides well-defined extension points for type-specific functionality. An example extension is to support looking up asset by path and checking for path overlaps, which is used by all asset types that have backing storage.

The data model is persisted in a standard relational database with the implementation detail hidden from the layers above. The layered design allows performance optimizations such as caching to be fully implemented within the persistence layer, as long as consistency guarantees are maintained. In Databricks, UC adopts a strong consistency model that enforces serializability of all metadata updates within a metastore. The details of persistence and caching are discussed in Section 4.5.

The adapter layer provides the integration with different asset types and with various cloud providers. To add an asset type to UC, developers add a declarative manifest to UC's asset types registry. The manifest is a specification of the asset type, including its location in the hierarchy, the operations and privileges supported on it, the authorization rules for each operation, and how its lifecycle should be managed. Developers can provide annotations or custom logic for validating the asset type's input attributes in Create, Read, Update and Delete (CRUD) APIs. For example, annotations can specify whether the `comment` field of a table securable is updatable, and if so, define the valid input length for that field. The adapter layer also defines a uniform interface for accessing cloud storage and handling cloud credentials, so that higher-level features can be implemented in a cloud-agnostic fashion.

The layer above implements core features critical for the catalog's integrity, such as namespace and lifecycle management, access control, storage path management and audit logging. The shared implementation ensures uniformity and correctness of core behaviors across all asset types. The core service exposes CRUD APIs for metadata management, credential APIs for engines to request temporary storage credentials, and a metadata query API that allows filter pushdown to support information schema functionality. It

also publishes metadata change events on asset changes, which is used for discovery catalog functionality, as described in Section 4.4.

4.2.3 Extending UC to be a MLflow Registry. This section describes using UC's entity-relationship model to add support for MLflow models and demonstrates extending UC to non-tabular assets. Unstructured data encompasses a variety of formats (e.g., text, images, audio, video), and the integration of MLflow models shows the ease of modelling unstructured data representations as UC asset types.

MLflow [34], like other ML frameworks such as Weights & Biases [33], includes a model registry: a centralized store to manage the full lifecycle of ML models and their versions, from staging to production. To add MLflow model support, UC was extended to be an MLflow model registry. This involved adding a new asset type called *model* to UC and extending the MLflow framework with modules to support the UC model registry.

In MLflow, a model registry consists of a concept called a *registered model*, which can have multiple versions. Representing these concepts in UC was straightforward. We created an asset type called `RegisteredModel`, which has multiple model versions. Most of the functionality to manage a `RegisteredModel` was inherited from the common functionality in the entity-relationship model's adapter layer, including organizing the model asset within the namespace (as a child of a schema), defining a CRUD API for models, defining the permissions required to access models, managing metadata storage, supporting cloud storage for model artifacts, enabling credential vending for cloud storage access, and implementing auditing, lineage tracking, and retention policies.

Extending the open-source MLflow framework to integrate with UC was also straightforward. This required implementing UC-specific versions of MLflow's base abstractions: `RestStore` and `ArtifactRepository`, which are the abstractions for a model registry REST endpoint and for the cloud storage storing the model artifacts. The UC `RestStore` implementation uses UC's registered model APIs to provide model registry functionality, and the UC `ArtifactRepository` implementation uses UC's model temporary credentials API to securely fetch credentials for reading or writing model artifacts in cloud storage.

4.2.4 Catalog Federation. Customers often have existing data in catalogs like HMS that they want to integrate into UC, with minimal effort and without redundant data copies, so that engines using UC can access the data under UC governance. This may be required either because data migration takes time or because the existing data is managed by a team separate from the one managing UC.

To support such scenarios and serve as a universal catalog, UC offers *catalog federation*. With federation, an administrator can create a federated catalog in UC that mirrors an existing *foreign* catalog by providing credentials to connect to the foreign catalog. The foreign catalog metadata is mirrored into the UC federated catalog either on demand during access or in the background.

In the current implementation, metadata mirroring occurs on demand. For example, when a query references a table in the UC federated catalog, the table's metadata is fetched from the foreign catalog and mirrored into the federated catalog. Similarly, when listing tables in a schema in the federated catalog, their metadata is mirrored. On-demand mirroring offers two key benefits: (i) it ensures that queries use the most up-to-date metadata from the foreign catalog, and (ii) it minimizes load on the foreign catalog by reusing the metadata fetched during query execution for mirroring.

Metadata mirroring can be performed either by the client engine or the catalog service. In the current implementation, this task is handled by the engine, as it avoids the need for customers to configure additional network or security settings to grant the catalog service (running in the Databricks control plane) access to the foreign catalog. The engine is typically configured to already have the required access to the foreign catalog. However, the tradeoff is that simple clients, like the UI, that only connect to the UC federated catalog and do not connect to the foreign catalog may see stale metadata until it is mirrored by some engine.

4.3 Uniform Access Control

This section describes how UC enforces uniform access control for securables regardless of whether they are accessed by name or path, and how it enforces access control at a finer granularity than a securable in collaboration with trusted engines.

4.3.1 Securable-level Access Control. The query flow shown in Figure 1 highlights the two places where access control is required: first, when accessing an asset's metadata, and second, when accessing its data. This applies universally to all asset types, though data access control isn't needed for asset types without associated data, such as catalogs and functions. UC centralizes the enforcement of both metadata and data access control within the Unity Catalog service. Metadata access control is enforced in the access path when a client calls the REST API to fetch metadata. The permissions required depend on the type of metadata operation—for example, administrator privileges are needed to change ownership, while MODIFY is sufficient to update a table's `comment` field.

UC does not enforce access control directly on cloud storage reads and writes, as it is by design not in the client's data access path to avoid performance bottlenecks. Instead, it controls access through a temporary credential vending mechanism. In this model, administrators grant storage access exclusively to the catalog service by configuring UC external locations and storage credentials, while clients only receive credentials to invoke UC and do not have direct access to cloud storage. When a client needs access to data, it invokes UC's temporary credentials API, specifying the required level of read or write access. If access is requested via a cloud storage path, UC resolves the path to a unique asset (according to the one-asset-per-path principle), validates the client's privileges, and issues a temporary credential scoped to the asset's storage path and access type. These credentials leverage the cloud provider's temporary credential system (e.g., STS tokens for AWS S3) and are valid for tens of minutes. While revoking privileges does not immediately block clients with active credentials, this trade-off is generally acceptable to customers seeking centralized access control.

The metadata and data access control mechanisms described here serve as building blocks for higher-level access control policies, including UC's current SQL grant-based permission model and the upcoming ABAC functionality.

4.3.2 Fine-grained Access Control. Some use cases require controlling access to specific rows and columns within a table. For example, when accessing an employee table containing salary information,

a user may be entitled to only the rows for employees who report to them. Table-level access control is insufficient in such scenarios. To address this, UC coordinates with the engine running the workload to employ a two-level access control mechanism that provides defense-in-depth: securable-level access control to grant the engine temporary credentials to access just the table, with fine-grained access control (FGAC) relying on the engine to enforce row filtering and column masking, based on user privileges. Securely applying FGAC requires an engine to be isolated from user code. Engines with this ability are called *trusted* (authenticated to UC with their machine identities) and access to tables with FGAC policies is restricted to only trusted engines. Engines that allow users to execute unsandboxed, arbitrary code are not trusted. In Databricks, sandboxing of user code from the core Apache Spark engine is implemented by the Lakeguard system [17] using containers.

FGAC principles also apply to views and shallow clones. In UC's governance model, granting SELECT privilege on a view or shallow clone allows access to its data, even if the user lacks privileges on its base tables. This requires enforcing access to specific subsets of the base tables' data, with the same FGAC principles and trusted engine restrictions applying for these use cases.

In some use cases, such as ML workloads requiring GPU access, the engine may lack the isolation needed for FGAC. To support access to tables with FGAC policies in such scenarios, UC supports a *data filtering service* [17], a trusted engine to which untrusted engines delegate queries involving FGAC policies. The data filtering service securely executes these queries and returns the results to the untrusted engines. In Databricks, an untrusted engine uses Spark Connect [5] to send the queries to the data filtering service.

4.4 Discovery Catalog Support

As Figure 3 shows, discovery catalog functionality like search and lineage is provided by second-tier services. However, they depend on the core service, and the interaction between them reflects a clear separation between "foreground" and "background" capabilities. The design balances latency, functionality, scale, and data freshness. Foreground capabilities, such as access control and audit logging, are integral to the core service, where ensuring low-latency responses and consistency is essential for the integrity and security of metadata. In contrast, background capabilities, such as search, discovery, and lineage, are built as extensions of the core service. They rely on platform-level features like metadata change events to asynchronously process metadata updates. This separation enables second-tier services to focus on large-scale indexing operations while tolerating slight staleness in metadata updates.

Metadata change events serve as the critical bridge between the core and second-tier services. Whenever metadata is modified, the core service propagates change events, which are consumed by second-tier services to update their indexes, graphs, or lineage models. This event-driven approach ensures background services stay synchronized with the core service while operating independently in terms of processing and storage. The decoupled design not only enhances scalability and fault isolation but also allows background services to leverage core platform capabilities without introducing significant complexity, ensuring a cohesive and scalable catalog that supports both operational and discovery catalog functionality.

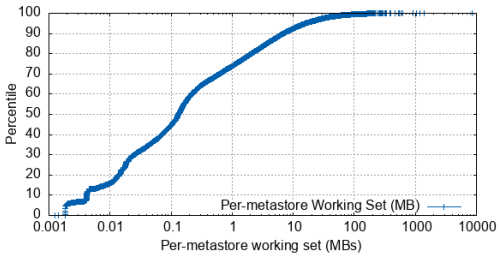


Figure 4: CDF of per-metastore working set sizes at steady state across all UC metastores, which shows that almost all metastores have working sets less than 100MB.

Another important functionality provided by the core service to second-tier services is an efficient authorization API. The second-tier services use this to govern the discovery catalog data. For instance, when a user performs a search, the search service uses this API to filter the results shown to the user.

4.5 Performance

As described in Section 3.4, UC is on the critical path for all workloads, including interactive analytics, BI tools, streaming, batch processing, and ETL jobs. Consequently, UC's access latencies directly affect the user experience across these workloads, and its availability and throughput determine the overall availability and performance of the workloads.

Several key characteristics of UC's workload enable it to meet its latency, throughput, and availability goals while supporting the required read and write semantics.

(1) UC stores and serves metadata, which is relatively smaller in size compared to the data. This makes it viable for UC to host the entire working set of metadata for a metastore or Lakehouse in memory. Moreover, UC's workload is predominantly reads, allowing it to use in-memory caching to provide low latency, scalable reads. Figures 4 and 5 support these points. Figure 4 shows that almost all current metastores have a working set less than 100 MB, while 90% have a working set of less than ~10 MB. Figure 5 demonstrates temporal locality—90% of container assets (e.g., schemas) across all metastores are re-accessed within 10 seconds of access. Similarly, 90% of leaf-level assets (e.g., tables) are re-accessed within 100 seconds. Together, they make the case for in-memory caching of UC asset metadata.
(2) All operations in UC are scoped to a metastore, and UC provides snapshot isolation for reads at a metastore granularity.
(3) UC provides serializable writes at a metastore granularity. Most writes are scoped to a specific asset and writes that span assets are relatively uncommon. Moreover, write APIs are also read-heavy because they perform various metadata validations and permission checks before the actual write. These characteristics enable UC to scale write throughput.
(4) UC's workload is especially amenable to batching, allowing callers to authorize and fetch the different metadata required for one or more operations in a single API call.

The following sections describe these aspects in more detail.

Caller-based optimizations. UC's workload is read-heavy, with ~98% of traffic being for read-only APIs that are used to fetch (i)

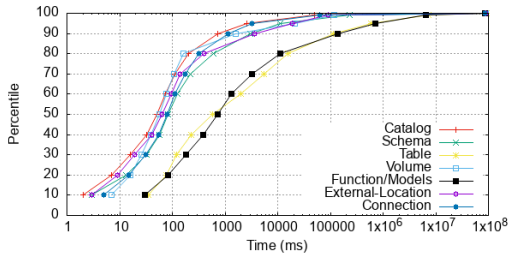


Figure 5: CDF of inter-arrival times of accesses for the same asset, plotted for different asset types. Container assets (catalogs, schemas) and dependencies of other assets (external locations, connections) have more frequent accesses (and hence lower inter-arrival times) than "leaf level" assets (tables, functions, models).

mutable asset metadata, for example, a table's name, columns, cloud-path, or permissions, and (ii) immutable metadata that UC obtains from other services, such as temporary storage credentials obtained from cloud provider APIs or metastore-region information obtained from UC peer services in other regions.

A single user operation, e.g., a SELECT query, typically requires both types of metadata, which UC serves in a single API call. Use cases where an asset depends on many other assets benefit significantly from such batching. A common example is nested views with several nesting levels that depend on 100s of base tables.

Immutable metadata is cached at multiple levels, both at the Unity Catalog service and at compute engines. In particular, the Unity Catalog service caches temporary storage credentials in its memory and on a RINK caching service [1] to survive restarts. It also allows engines to cache them for the period of their validity (typically 10s of minutes) and reuse them (when permissible) across successive queries or across multiple Spark executors.

Mutable metadata caching. UC uses an ACID-compliant database (DB) as its backend, and implements a write-through in-memory cache to optimize reads while guaranteeing metastore-level snapshot reads and serializable writes. The cache's design has two key components. First, UC shards metastores across its nodes, with each node *owning* one or more metastores and responsible for caching them. Metastore-to-node assignments can be static or dynamic, and UC does not assume exclusive ownership.

Second, UC uses metastore versions to provide reads with snapshot isolation and writes with serializable isolation, and to detect when more than one node owns a metastore and take remedial action. This design avoids reliance on distributed-consensus-based services, like ZooKeeper [23], for consistency. The metastore versions are persisted in the DB and a node owning a metastore also caches the version in memory. Assets within a metastore don't have persistent versions; they only have an in-memory version, which is the version of the containing metastore when an asset is read from the DB into the cache. Versioned assets in the cache are identified by their primary and secondary keys, for example, a table's cached metadata can be retrieved using either its ID, its path, or its name. To allow in-progress reads to not block concurrent writes, the cache is multi-versioned, but minimizes versions by maintaining only the most recent asset versions.

To ensure snapshot reads, a node owning a metastore maintains the invariant that a cached asset's versions are the latest as of the metastore version known to the node. To ensure this, on every DB read, the node uses the DB's metastore version to check that its in-memory version is the latest one. Otherwise, it *reconciles* the cache by bringing it up-to-date with the DB. A naive reconciliation strategy is to evict all cached-state for a metastore, while an optimized strategy is to consult a change-event system to selectively invalidate cached entries that were modified between the cached version and the version in the DB.

To provide serializable writes, a write to an asset increments its metastore version in the DB, conditioned on it being the current in-memory version. If this fails, another node could have written to the metastore and the node initiates reconciliation as explained earlier. If the write succeeds, the new asset version is inserted into the cache to maintain the invariant mentioned above.

Cache eviction. UC's multi-version cache requires two types of eviction. First, to limit memory consumption of unpopular assets, we use standard eviction algorithms, such as LRU and LFU, to evict an unpopular cached asset and all its versions. Second, to limit the number of cached versions for popular assets, we use the timeout enforced for every UC API call by an upstream load-balancing proxy. The idea is that when a write to an asset adds a new version to the cache, existing cached versions of the asset will be in use by in-flight requests for at most a timeout period of time. Past that, existing versions can be evicted and it is done lazily by the next request accessing the asset. Together these eviction mechanisms limit the versions of popular assets and page out unpopular ones.

5 Implementation

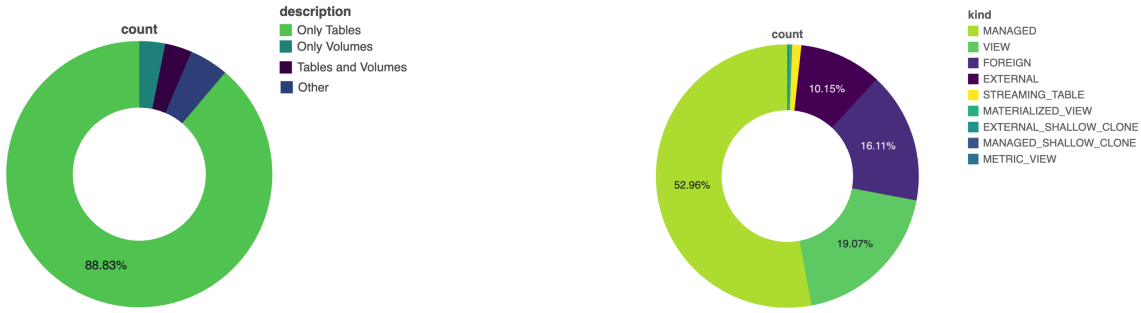
Databricks UC is implemented as a web service in Java and Scala, and an open-source Java-only version without dependencies on Databricks internal services is available at https://www.unitycatalog.io/. UC supports using various OLTP databases as the backend. It is deployed in multiple regions in Azure, AWS, and GCP, and has a variety of clients including engines like Apache Spark, UIs like Databricks Catalog Explorer, and external tools like Immuta, PowerBI, and SQLAnalytics.

Databricks UC servers are sharded using an internal sharding service that, similar to Slicer [2], provides best-effort metastore-to-node assignments with no hard guarantees. UC's mutable metadata cache is extensible to support different in-memory indexing structures optimized for different read patterns, and it currently uses hash-maps, versioned-lists and URL-tries. These allow UC to efficiently serve point lookups for assets (e.g., details of a table, schema, or catalog, by name or ID), privileges, memberships (e.g., list of models in a schema), as well as complex reads (e.g., finding assets with storage paths overlapping with a given path, which is used during asset creation or credential authorization).

6 Unity Catalog in Practice

This section reports on UC's real world customer usage and performance, and shows that they support the need to address the challenges outlined in Section 1. In particular, we find that:

- Assets per catalog follows a typical heavy-tailed distribution.



(a) Usage of tables, volumes, and other types, measured by counting schemas that have only tables, only volumes, only tables and volumes, and other types in addition to tables and volumes.

(b) Distribution of the number of tables of different types.

Figure 6: Distribution of different asset types used by customers, which supports the need for UC to support diverse asset types.

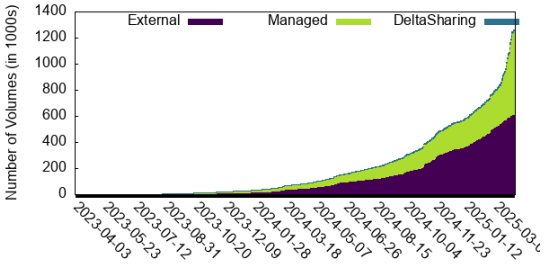


Figure 7: The growth in the number of volumes created is accelerating over time.

- Tables are accessed by both catalog name and storage path, highlighting the need for uniform access control.
- Customers use a variety of asset types and different table formats, and the accelerating usage of non-tabular assets underscores the need to support diverse asset types.
- The variety of external BI tools accessing UC data and the growth of Delta Sharing external recipients show the need to support external access.
- UC's current performance is on par with that of local HMS, in spite of UC being a remote service and the significant additional functionality of UC over HMS (e.g., governance), demonstrating the benefits of performance optimizations from Section 4.5.
- UC enables use cases not possible before, with predictive optimization showing a 20× query latency improvement.

6.1 Aggregate Usage Statistics

Currently, UC serves ~9,000 distinct daily active customers, across more than 75K workspaces. This generates ~60K API requests per second to UC across all regions, with 98.2% of requests being reads and the remaining ~2% writes. These API calls support various operations such as creating, updating, and deleting asset types, as well as managing access policies and grants.

UC governs more than 100M tables, 550K volumes, 400K models across 4M schemas, 200K catalogs, and 100K metastores. Assets of each type in UC follow a heavy-tailed distribution, as is common in practice. Many catalogs contain only a few assets and the mode of the distribution of volumes per catalog is fewer than six. This suggests that volumes are often used as directories for files, with a handful per catalog sufficing in many cases. For tables, the mode is ~30 tables per catalog. At the tail of the distribution, the largest catalogs by table count contain ≥ 500K tables each, and the largest catalogs by volume count have several thousand volumes each. UC's architecture flexibly supports this wide range of catalog sizes to enable workloads that represent both the head and the tail of this distribution—for customers with millions of assets in one catalog, metadata handling itself turns into a "big data" challenge.

6.2 Evidence for Unity Catalog's Key Challenges

We use real-world customer usage of UC's functionality to demonstrate the importance of the key challenges laid out in Section 1.

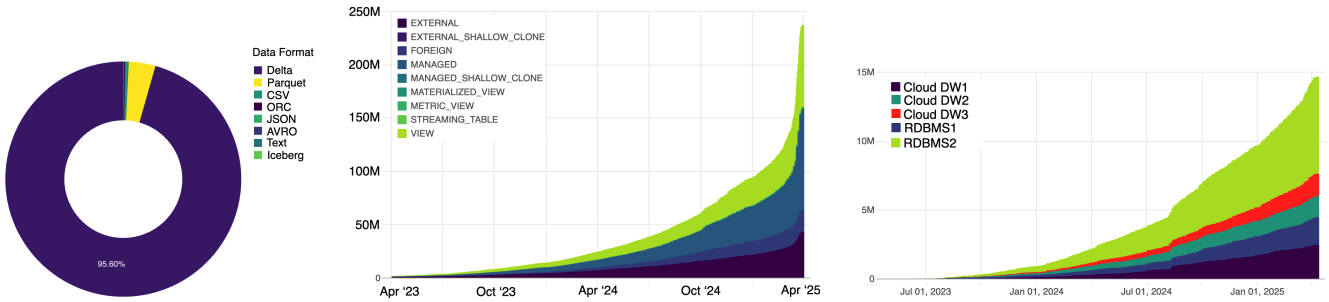
The need for uniform access control. Figure 11 shows the fraction of tables that are accessed using their catalog names, using their cloud storage paths, or both. Though most of the tables are only accessed using their catalog names (as expected), ~7% of tables are also accessed using their paths. This highlights the need to support path-based access for tables, and consequently the need for uniform and consistent access control whether a table is accessed using its catalog name or storage path.

The need to support diverse asset types. UC supports diverse asset types beyond tables, and within tables supports multiple table types (e.g., managed, external, and views) and storage formats (e.g., Delta, Iceberg, and Parquet). We observe that real customer usage highlights the importance of supporting these diverse asset types.

Figure 6(a) shows the relative usage of various asset types, focusing on tables and volumes, with other types grouped under "Other". While ~89% of schemas contain only tables, ~3% contain only volumes, ~3% contain both tables and volumes, and the remaining ~5% include a mix of asset types (e.g., and ~2% contain only ML models, which existing tables-only catalogs do not support).

Additionally, Figure 7 shows that the creation of volumes is accelerating over time, indicating that non-tabular asset types like volumes (which store unstructured data) and ML models will continue to grow in importance as AI/ML workloads expand in enterprises.

Within tables, Figure 6(b) shows the usage distribution for different table types: managed tables account for ~53% and are the most common but other table types also have significant adoption. Furthermore, Figure 8(b) shows the growing usage of all table types,



(a) Distribution of table storage formats. Delta is the dominant format.

(b) Distribution of different table types over time.

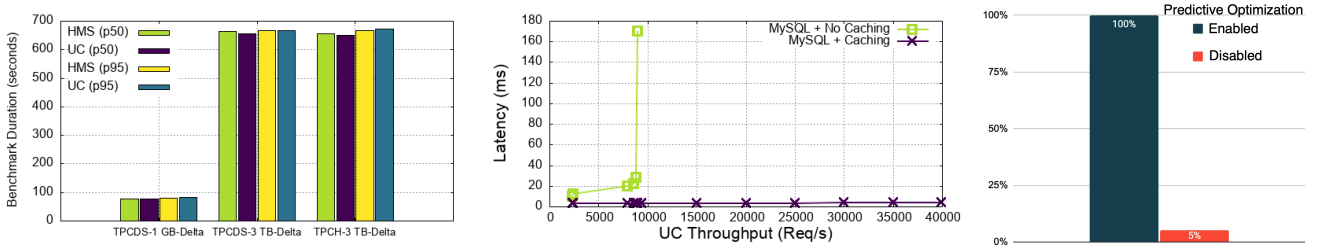
(c) Distribution of the top five foreign table types over time.

Figure 8: Tables are the most used asset type, which are themselves a diverse group of types and storage formats.

(a) External clients calling UC.

(b) External clients calling HMS.

Figure 9: External clients (y-axis) calling UC and HMS over a 14 day period, and the type of SQL queries they run (x-axis). A bubble's size represents the number of queries of a type run by external clients of a given type. The x-axis and y-axis do not list all commands and all external clients for brevity.



(a) Performance comparison of HMS vs. UC for TPCDS and TPCH benchmarks. HMS is in "local metastore" mode, which is its best-case configuration.

(b) Benefit of UC caching on latency and throughput.

(c) Query latency improvement due to predictive optimization, enabled by UC's metadata management.

Figure 10: Performance of UC in practice.

underscoring the need for broad support. In contrast, HMS supports only managed tables, external tables, and views, which cover 82% of table usage and 67% of usage of all asset types.

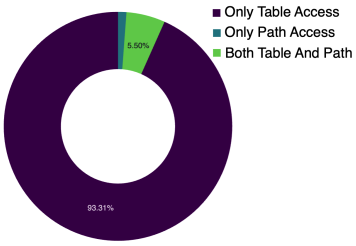


Figure 11: Distribution of number of tables with accesses via only the table's catalog (shown in purple), only its cloud storage path (shown in blue), or both (shown in green).

The need to support multiple table formats is showcased in Figure 8(a). While the majority of tables are in the Delta format, customers also use other storage formats. Moreover, Figure 6(b) shows that ~16% of tables are foreign tables; UC currently supports 26 foreign table types and Figure 8(c) shows the increasing usage of the top 5 foreign table types, of which three are from other well-known cloud data warehouses. This demonstrates how federation helps UC support diverse table formats and be more open.

Volumes are the most common non-tabular asset type, which are used as containers for files for a variety of use cases. From anecdotal evidence, these use cases include: storing large collections of unstructured data (e.g., images, audio, video, text, PDFs) for AI/ML workloads; uploading and querying non-tabular files for data exploration; working with tools lacking native support for cloud object storage APIs (and expect files in the cluster's local file system); and staging and pre-processing raw data files during early stages of ingestion before loading them into tables. These are critical use cases that highlight the need to support non-tabular assets—capabilities missing in existing catalogs like HMS.

The need for external access is perhaps harder to show with usage data as most workloads use the Databricks Runtime. However, we observe a large diversity of external clients that call UC and we contrast this with external clients that call HMS (for customers using HMS). Figure 9 shows this comparison: the number of external client types that call HMS is 95 (Figure 9(b)), which is ~3.5× smaller than the 334 client types that call UC (Figure 9(a)). Additionally, a broader range of query types are invoked on UC, with 90 types compared to HMS's 30. Many of these clients and tools used by our customers are unknown to us, which emphasizes the need for openness to make such integrations work well.

Another indicator of the need for external access is Delta Sharing [15]. Databricks UC implements the open Delta Sharing protocol to allow customers to easily share data with recipients internal or external to Databricks. This feature has seen wide adoption, and usage metrics show that external data sharing recipients contribute to a larger share of usage than internal recipients.

Performance of UC in practice. We evaluate UC's end-to-end performance using TPC-DS [26] and TPC-H [10] benchmarks. Tables are in the Delta storage format, with UC configured to use an AWS db.m5.24xlarge MySQL instance as the backend database and the optimizations described in Section 4.5 enabled. We compare it to performance on HMS using a AWS MySQL instance of the same size for its metastore DB configured as a "local metastore" [22], where engines use JDBC to directly make SQL queries to the metastore DB. This setup represents the optimal configuration for HMS. In contrast, UC's architecture is more similar to HMS's slower "remote metastore" setup, where engines communicate with the metastore over a RPC interface, which introduces additional latency.

Figure 10(a) shows that there is no statistical difference between the performance of UC and HMS, in spite of UC being a remote metastore and providing extra capabilities not offered by HMS, including privilege enforcement and temporary storage credential generation. The UC optimizations from Section 4.5 play an important role in UC's competitive performance even when handicapped. Figure 10(b) quantifies the benefit of UC's server caching—it shows the latency vs. throughput for a sample API used in the query path, under different client loads with the same AWS MySQL instance as the DB. Caching significantly boosts UC's performance, with 3× to 40× lower latency while scaling to higher request throughputs. Without caching, the system is bottlenecked by database reads and reaches its throughput limit at fewer than 10K requests per second.

6.3 New Applications Enabled by Unity Catalog

UC enables several important new use cases at Databricks, of which we highlight two here.

Predictive optimization [30] is a recent feature that automates key maintenance tasks such as optimizing data file layouts, removing unused files, performing incremental clustering, and updating statistics. This removes the toil of manual maintenance and is enabled by UC's metadata management. As shown in Figure 10(c), predictive optimization significantly improves query latency: for a TPCDS data set with 1M rows, it reduces the latency of a query selecting ~5% of the rows by up to 20×. This gain comes from optimizing table file sizes using metadata stored in UC. Additionally, predictive optimization's garbage collection of unused files improves storage efficiency by up to 2×.

Multi-table and multi-statement transactions. While ACID table formats like Delta Lake support single-table transactions by relying on storage layer atomic operations, extending this to multi-table and multi-statement transactions is more complex as they involve updates to metadata and data of multiple catalog objects that can be stored on different storage buckets. As the centralized metadata store, UC plays a critical role in enabling such transactions via the ongoing work on Catalog-owned Delta tables [12].

7 Conclusion

This paper introduces Unity Catalog, an open and universal Lakehouse catalog developed at Databricks to address the limitations of existing catalogs. Unity Catalog supports a wide range of asset types, including tabular, unstructured, and AI assets, and integrates with various engines through an open API, while being multi-cloud. It provides consistent governance across asset types, engines, and clouds, interoperates with external catalogs, and delivers high performance. We discussed the challenges in building Unity Catalog and how it addresses them, and validated its goals and approach using real customer usage data.

References

[1] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 113–119.

[2] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer: Auto-Sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 739–753.

[3] Apache Iceberg. 2025. https://iceberg.apache.org/.

[4] Apache Polaris. 2025. https://polaris.apache.org/.

[5] Apache Spark Connect overview. 2025. https://spark.apache.org/docs/latest/spark-connect-overview.html.

[6] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Świtakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta Lake: High-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[7] AWS Glue. 2025. https://docs.aws.amazon.com/glue/latest/dg/what-is-glue.html.

[8] Philip A. Bernstein. 2003. Applying model management to classical meta data problems. In *First Biennial Conference on Innovative Data Systems Research, CIDR 2003, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*. www.cidrdb.org. https://courses.cs.washington.edu/courses/csep544/04sp/lectures/bernstein03.pdf

[9] BigLake. 2025. https://cloud.google.com/biglake.

[10] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391*. Springer-Verlag, Berlin, Heidelberg, 61–76. https://doi.org/10.1007/978-3-319-04936-6_5

[11] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther Hagleitner. 2019. Apache Hive: From MapReduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1773–1786. https://doi.org/10.1145/3299869.3314045

[12] Catalog-owned Delta tables. 2025. https://github.com/delta-io/delta/issues/4381.

[13] Dataproc Metastore. 2025. https://cloud.google.com/dataproc-metastore/docs/overview.

[14] Delta Lake deletion vectors. 2025. https://docs.delta.io/latest/delta-deletion-vectors.html.

[15] Delta Sharing Procotol. 2025. https://github.com/delta-io/delta-sharing/blob/main/PROTOCOL.md.

[16] Delta UniForm (Universal Format). 2025. https://docs.delta.io/latest/delta-uniform.html.

[17] Martin Grund, Stefania Leone, Herman van Howell, Sven Wagner-Boysen, Sebastian Hillig, Hyukjin Kwon, David Lewis, Jakob Mund, Xiao Li, Polo-Francois Poli, Lionel Montrieux, Othon Crelier, Michalis Petropoulos, Thanos Papathanasiou, Reynold Xin, and Matei Zahari. 2025. Databricks Lakeguard: Supporting fine-grained access control and multi-user capabilities for Apache Spark workloads. In *Proceedings of the 2025 ACM SIGMOD International Conference on Management of Data (SIGMOD '25)*. Association for Computing Machinery, New York, NY, USA.

[18] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795

[19] Rihan Hai, Sandra Geisler, and Christoph Quix. 2016. Constance: An intelligent data lake system. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 2097–2100. https://doi.org/10.1145/2882903.2899389

[20] Alon Halevy, Flip Korn, Natalya F. Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing Google's datasets. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 795–806. https://doi.org/10.1145/2882903.2903730

[21] Joseph M. Hellerstein, Vikram Sreekanti, Joseph E. Gonzalez, James Dalton, Akon Dey, Sreyashi Nag, Krishna Ramachandran, Sudhanshu Arora, Arka Bhattacharyya, Shirshanka Das, Mark Donsky, Gabriel Fierro, Chang She, Carl Steinbach, Venkat Subramanian, and Eric Sun. 2017. Ground: A data context service. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p111-hellerstein-cidr17.pdf

[22] Hive Metastore administration. 2025. https://cwiki.apache.org/confluence/display/Hive/AdminManual+Metastore+Administration.

[23] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

[24] Iceberg REST Catalog API. 2025. https://github.com/apache/iceberg/blob/main/open-api/rest-catalog-open-api.yaml.

[25] Phokion G. Kolaitis. 2005. Schema mappings, data exchange, and metadata management. In *Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Baltimore, Maryland) *(PODS '05)*. Association for Computing Machinery, New York, NY, USA, 61–75. https://doi.org/10.1145/1065167.1065176

[26] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 1049–1058.

[27] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data lake management: Challenges and opportunities. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1986–1989. https://doi.org/10.14778/3352063.3352116

[28] OneLake, the OneDrive for data. 2025. https://learn.microsoft.com/en-gb/fabric/onelake/onelake-overview.

[29] Postgres Catalog. 2025. https://www.postgresql.org/docs/current/catalogs.html.

[30] Predictive optimization for Unity Catalog managed tables. 2025. https://docs.databricks.com/en/optimizations/predictive-optimization.html.

[31] Snowflake Horizon Catalog. 2025. https://www.snowflake.com/en/data-cloud/horizon/.

[32] Trino: a query engine. 2025. https://trino.io/.

[33] Weights and Biases. 2025. https://wandb.ai/site/.

[34] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. 2018. Accelerating the Machine Learning lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45. http://sites.computer.org/debull/A18dec/p39.pdf

[35] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[36] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A unified engine for big data processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664